A Framework to Support Multiple Security Policies

Winfried E. Kühnhauser and Michael von Kopp Ostrowski
German National Research Centre For Computer Science (GMD)
Schloss Birlinghoven
D-53754 Sankt Augustin, Germany
email: winfried.kuehnhauser@gmd.de

Abstract

For many years the traditional concept of the reference monitor has proven to be a sound architectural foundation for secure computer systems. However, with the emergence of complex distributed systems with multiple, user-defined security policies the limitations of reference monitors become more and more obvious.

In the age of application-dependent, user-defined security policies, it is particularly obvious that the scope of the reference monitor concept is restricted to access control policies. Several security policies of modern application systems address much broader aspects of system security, such as authentication policies or guarantee-of-service policies.

In the age of multiple coexisting security policies in large systems, it is particularly obvious that the strong encapsulation property of the reference monitor concept makes it conceptually difficult to integrate new policies (or modify existing ones) into the trusted computing base.

This paper discusses the general concepts of a formal framework for supporting security policies in a multipolicy environment. The framework provides a library for security policies and policy implementation support via mechanisms for policy separation, policy persistency, communication, and reusability. Moreover, it provides a view for a formal foundation for reasoning about policy equivalence, policy cooperation and policy conflicts.

Introduction

In the last decade, the ever increasing amount of software application systems has confronted us with higher and more individual security requirements. Security policies have been developed that support the needs of individual applications, with

the Bell/LaPadula model (providing confidentiality) being only the first in an ever-increasing multitude [BL76, Bib77, CW87, BN89, San92a]. As a consequence, the integration of application dependent security policies into a distributed multipolicy computer system has become a major challenge for computer security.

Several security policies in modern application systems address a wide variety of aspects of system security, such as access control, authentication, availability, and auditing. This variety requires concepts in the underlying system platform that go far beyond traditional access control mechanisms. Although the traditional architectural concept of the reference monitor as defined in the Trusted Computer System Evaluation Criteria [Dep83] has been a successful principle within its intended scope, its power for supporting a wide range of application-specific policies is limited.

This paper is a contribution to the field of security engineering. It describes a formal framework that aims at supporting multiple application-specific security policies in distributed systems. While the framework provides formal concepts for defining security policy properties, the mechanisms that come along with the framework aim at closing the gap between a high-level security policy and its integration into a system’s trusted computing base. Once a security policy is developed and translated into a security model within the framework, the framework supports

- **Policy Coexistence** in multipolicy systems by providing a foundation for reasoning about policy conflicts and cooperation of adjacent security policies through a formal base

- **Policy Reusability** by promoting a strict orthogonality between security policy and application, providing a policy library and a base for reasoning about policy equivalence, concatenation, or ranking. Reusability is the base for transporting all the work that has already been spent on security models into a multipolicy environment

- **Policy Implementation** by providing a common set of security mechanisms, policy persistency, policy communication, and the three traditional reference monitor principles of tamperproofness, total access mediation, and modularity.

We apply our ideas to distributed systems consisting of connected, autonomous and mutually suspicious computer systems\(^2\). We also assume that within such a system several independent application systems coexist. The individual security requirements of these applications are reflected in application-specific security policies associated with the applications. Whenever the applications and their security policies interact, metapolicies coordinate the involved individual security policies.

\(^2\)The Network Interpretation [Nat87] of the Trusted Computer System Evaluation Criteria [Dep83] calls this view of a computer network the "multiple interconnected systems view", in contrast to the "single trusted system view"
Application systems are made up of interacting entities. Any entity interaction is mediated by local and autonomous reference monitors. More precisely, a communication scheme is assumed that exactly restricts communication to sending and receiving messages; reference monitors control both the sending of a message at the sending entity and the reception of a message at the receiving entity. On top of this basic communication scheme, higher level paradigms may be found, including, but not limited to, a remote procedure call communication scheme or a client/server interaction paradigm.

Thus the general context of the paper are systems consisting of

- independent, interconnected and mutually suspicious hosts with
- application systems controlled by application-specific security policies and
- security policies controlled by metapolicies.

**The Framework**

Success in achieving a high degree of security in a computer system depends on the degree of care put into designing and implementing the security-related software parts. The foundation of a system's security is the security policy, containing all security-related requirements. A security model is a precise representation of a security policy and thus is the starting point of any security policy implementation.

The guiding principles in the design of our formal framework are:

- We aim at providing a simple understandable formal framework that is applicable to a wide variety of security policies.

- A large and complex system generally has several applications and security policies in place. The framework provides a foundation for reasoning about policy conflicts and cooperation, a concept usually referred to as metapolicies [Hos92].

- The amount of effort invested in software development makes it generally desirable to reuse software in a variety of applications. Security policies and their formal models are no exception. On the contrary, because of their need for completeness and correctness, security models belong to a very expensive class of software. It is thus highly desirable that a security model, once developed, coded, and verified, is used for more than a single application.

- The framework supports the transition from the formal model into a program module and its integration into the trusted computing base via the custodian paradigm that mechanically provides policy tamperproofness, policy separation, policy persistency, and policy communication [Küh95].
Our first principle states that we are aiming at a formalism that is applicable to a wide variety of security models. We recognize that today we are looking at many families of security models, e.g. the information flow family, the algebraic family, or the noninterference family. Forcing all models into the same formalism generally antagonizes the expressive power of the individual models.

Consequently, our aim is a framework for families of security models, each family supporting security models written in terms of the family language. The framework is open for new families, thus providing support for evolutionary progress in the security model area, such as the work of W. Sandberg on an Open Reference Model for security policies [SM94] implies.

Figure 1: The Framework Architecture

The second principle states that in a multipolicy environment, there are generally several applications in place, and generally, each application has its individual security policy. Interaction of the applications then involves more than a single security policy. This raises the need for defining rules for policy cooperation and conflict handling at the policy perimeter, a concept usually referred to as metapolicies.

While each model in the framework is written in the terms of the family language, the framework additionally provides a common, family neutral foundation. This foundation is both a habitat for security policies and the information base for metapolicies. As a habitat, the common foundation contains a policy library as well as mechanisms supporting policy implementation and providing reference monitor properties for user-defined policies, policy encapsulation, persistency, and communication. As an information base for metapolicies, the common foundation contains a formal base for defining inter-policy relationships.

The current work on the framework is focused on supporting three model families: the family of algebraic models, the family of lattice-based models, and the family of expert system based models. Within the framework, a policy implementation is a formal model representation, consisting of a high level semantical description (in terms
of algebras, lattices, or expert system rules and facts) and a model implementation in terms of executable code. A single policy may have more than a single model representation; the examples at the end of the paper shows an algebraic and an expert system rules & facts based representation of the Chinese Wall security policy. A lattice-based representation can be found in [San92a].

The following section provides a brief description of the three currently supported model languages. The major part of that section then concentrates on the family neutral common foundation.

The Model Families Level

The Algebraic Family

Algebras have long since proven to be a successful concept for specifying software systems and reasoning about their correctness. An algebra \( \mathcal{A} \) consists of a set of types \( T \), a set \( O \) of operations on these types, and optionally a set of axioms. The pair \( (\Sigma = (T, O)) \) is called the signature of \( \mathcal{A} \). \( \Sigma \) together with a set of axioms is called a specification.

The experience with algebra has resulted in many security policies that are specified by an algebraic model. In the framework, an algebraic model is a signature (defining the syntax of a specification) and has a non-empty set of axioms. The axioms are interpreted according to the initial semantics approach [GTW78] in contrast to the final semantics approach [Wan79]. Every security policy defines an interface that may be used by applications, other security policies or metapolicies to communicate with the policy. This interface is the only component in the framework’s algebraic family that goes beyond the strict mathematical definition of an initial algebra.

While this approach supports the functional specification of a security policy, complex policies require additional concepts for decomposition and composition. The algebraic family provides modules for separate specification of abstract data types, generic modules for type-independent specification of abstract data types (like arrays, lists, stacks), and inheritance of complete security policies.

The Lattice Family

The lattice has been traditionally used in computer security as a mechanism for access-control models to control the flow of information. This was done by attributing all the entities of the system being governed by the policy with an entry from a partially ordered set. Since legal information flows between entities attributed with different set entries are, by nature, transitive, reflexive and anti-symmetric, there is a natural mapping between the information flow security constraints and the lattice structure.

We are looking at the lattice-based family not only because the many existing information flow models exhibit the lattice properties, but also because many more
access control models can be expressed using the lattice. We have shown that any access control model which associates attributes with the entities being governed can be re-written in the form of a lattice [Bry95]. We have also defined an algebra of lattice-based policies, allowing lattice-based policies to be compared and combined using a Boolean algebra. The uniform lattice structure of these policies helps to make policy combination mechanical.

**The Expert System Family**

Policy making is a human enterprise; security policies and models rarely are perfect from the start. Changes to existing and verified formal models are expensive and generally require expert knowledge.

Evolution, learning and rapid prototyping are domains of artificial intelligence. The framework uses expert systems to support security policy evolution and rapid prototyping. Models within the expert system family are written in terms of rules and facts of the Mobal expert system [MWKE93]. The direct interpretation of the rules and facts by the expert system provides the model developer with immediate practical consequences of any changes to the rules/facts base.

After being satisfied with the model behaviour, the rules and facts constitute the starting point for a formal model analysis. This aspect of the framework is ongoing research. However, the paper will give an example that demonstrates a working model of the Chinese Wall security policy in terms of Mobal rules and facts.

**The Common Foundation**

The goal of the common foundation is to support policy coexistence, reuse and implementation as well as to provide a foundation for defining inter-policy relationships in multipolicy environments. One of the major future goals is the definition of an algebra that supports policy comparison and policy synthesizing.

The common foundation has two major components: it is an *information source* for reasoning about policy cooperation and conflict handling by metapolicies, and it is a *habitat* for security policies.

**The Policy Habitat**

As a habitat, the common foundation supports policy reuse via a policy library and policy implementation via the custodian paradigm. Maintaining security policies in a policy library requires the strict separation of security policy and application. To glue together a policy from the library and an application, the custodian paradigm provides the necessary glue by providing total communication mediation, a principle well known from the traditional reference monitor concept.
Each policy in the library is implemented by one or more security models, written in the language of a model family. Each security model in turn is implemented by one or more custodian types (or classes), written in a regular programming language. An instance of a policy is created by instantiating one of its custodian types. Figure 2 shows the three different abstraction levels of a policy within the framework. The figure shows the Chinese Wall policy implemented both in the algebraic and in the expert system family, the algebraic model implemented in two classes in different programming languages, and the C++ custodian having two instances.

![Policy Representation Within the Framework](image)

Figure 2: Policy Representation Within the Framework

To provide a basis for reasoning about policy cooperation and conflict handling, a few formal definitions of the habitat structures are now in order. Based on our principle of providing a simple and broadly understandable framework, we use only simple elements from set theory: sets, functions, and relations.

**The Policies Level.** On the policies level, each policy has a unique identifier, a type, an interface definition, an implementation, and a set of certificates.

The policy identifier is the foundation for policy authentication and for defining policy relations like ranking or semantic equivalence. We use the letter $\mathcal{P}$ to denote the set of all security policies within the framework.

The type of a policy is defined by a bundle of operations common to all policies of the same type (such as "Check Access" in case of access control type policies). The set of types is called $\mathcal{T}$, and

$$
type : \mathcal{P} \rightarrow \mathcal{T}
$$

is the corresponding function that results in the type attribute of a given policy. Examples for types are authentication policies, access control policies, and audit policies.
The interface definition describes the operations or methods that are exported by a policy. Besides the type-dependent operations, each security policy has an individual interface. As an example, in the Chinese Wall policy the creation of a new branch within a company is modelled by an operation that sets two formerly unconflicting companies into conflict. Because of their general success in software specification, algebraic signatures are used for specifying policy interfaces.

The set of all formal security models within the framework is called $\mathcal{FM}$. The implementations section maintains all those security models that implement the policy.

<table>
<thead>
<tr>
<th>PolicyName:</th>
<th>$\mathcal{P}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolicyType:</td>
<td>$\mathcal{T}$</td>
</tr>
<tr>
<td>PolicyInterface:</td>
<td>algebraic signature</td>
</tr>
<tr>
<td>PolicyImplementations:</td>
<td>SET OF $\mathcal{FM}$</td>
</tr>
<tr>
<td>Certificates</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: The Policies Level

The Models Level. Each security policy is implemented by one or more security models. Each security model is described by a family, a formal representation in terms of the family language, and one or more implementations derived from the formal representation.

Within the same family of models and based on the common family language, the framework supports operations such as formal model equivalence (an important feature for policy reuse). The set of families is called $\mathcal{F}$, and

$$family : \mathcal{FM} \rightarrow \mathcal{F}$$

is the corresponding function that results in the family attribute of a given model. Current valid families are the algebraic family, the lattice family, and the expert-system-based family.

The formal representation of a security model is a precise definition of the semantics of the security policy and is the basis for reasoning about model equivalence. The set of all formal representations is called $\mathcal{M}$, and

$$model : \mathcal{FM} \rightarrow \mathcal{M}$$

is the corresponding function that results in the formal representation of a given model.

The set of all formal custodian classes within the framework is called $\mathcal{CC}$. The implementations section maintains all those custodian classes that implement the model.
in terms of a programming language. An instance of a policy thus is created by instantiating a custodian class.

**The Policy Instances Level.** Each security model is implemented by one or more custodian classes. Each custodian class can be instantiated to obtain a new security policy instance.

Each security policy instance has a perimeter definition in time and space, consisting of a time interval between `start` and `expire date` and the `domain` that the policy controls.

The set of domains is called $\mathcal{D}$, and

$$\text{domain} : \mathcal{CC} \rightarrow \mathcal{D}$$

is the corresponding function that results in the domain attribute of a given policy. Domains may be defined based on several classification criteria. Domains may be **hierarchical**, reflecting the departments and sub_departments within a large organization; a hierarchical domain is usually contained in another, higher-level domain. A second classification criterion is the **ownership** of objects; objects belonging to the same organization usually constitute an ownership domain. Ownership domains typically do not overlap. Last but not least, domains may be **wild** or **task-oriented**. This situation generally occurs for groups of objects that are commonly accessed by cooperating projects from different organizations in open tele-cooperation systems.

The framework does not prescribe any particular domain concept or implementation: concepts as described in [SM89, Küh94] may be adopted.

The transition from the formal model into an immediate practical usability is supported by the custodian paradigm [Küh95] which has been integrated into the common foundation of the formal framework. Custodians

- sustain the reference monitor properties tamperproofness, total communication control, and formal verifiability,

- support the dynamic integration of user-defined security policies into the trusted computing base of the underlying system platform.
A custodian is an extension to a policy neutral reference monitor that encapsulates a user-defined security policy. A custodian provides policy integrity and separation, policy persistency and communication. The paradigm emphasizes the separation of policy and application, linking both via a binding mechanism within the underlying reference monitor. In terms of the reflective object model [YTM+91], a custodian provides the security part of an application object’s metaspace.

Custodians have been implemented before within the BirliX Security Architecture [HKK93] and within the Distributed Computing Environment of the Open Software Foundation [Fou92].

A policy instance thus is created by instantiating the custodian type that contains the formal model’s executable code. The instances of a policy are maintained together with instance-dependent information: dates controlling the effectiveness period of an instance, a certificate for the installation correctness. We use $\mathcal{CI}$ to denote the set of all security policies instances.

<table>
<thead>
<tr>
<th>InstanceDomain: $D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartDate</td>
</tr>
<tr>
<td>ExpireDate</td>
</tr>
<tr>
<td>InstanceImplementation: $\mathcal{CI}$</td>
</tr>
<tr>
<td>Certificates</td>
</tr>
</tbody>
</table>

Figure 5: The Policy Instances Level

Certificates. Certificates on all levels identify people or organizations that are responsible for the entries of the corresponding level. Metapolicies use certificates to judge on the trustworthyness of policy attributes and implementations.

The Metapolicy Support

As of today, metapolicies are an unproven concept. There has been little research into the concept, and we do not yet have a rich set of pragmatic examples. It is generally agreed that, while security policies control the interactions of application system objects, metapolicies are policies about security policies and thus control security policy interactions.

Within the framework habitat, a metapolicy is treated like any ordinary security policy, except that the objects within its domain $D$ are security policies instead of application system objects. Because of the novelty of the metapolicy concept we feel that it is yet too early to decide on an fully defined algebraic formalism for dealing with policy comparison and policy combination. Thus the second major component
of the common framework, the metapolicies support, concentrates on two elementary concepts: the conflict matrix and the cooperation matrix.

**The Conflict Matrix.** The conflict matrix handles situations where more than a single security policy instance is involved in a decision. To that end, the framework contains a ranking between policy instances via the relation

\[ \text{dominates} \subseteq CI \times CI \]

\text{dominates} defines a partial order on the policy instances. If for two policy instances \( i_k, i_l \in CI \) the relation \( i_k \text{ dominates} i_l \) holds, then in case of conflicts, a metapolicy can select the decision of the dominating policy without external assistance.

The \text{dominates} relation is represented within the framework by the more general conflict matrix \((CI \times CI) \rightarrow \text{CRF}\) that maps a pair \( i_k, i_l \) to a conflict resolution function. Valid conflict resolution functions are - as the above example - the \text{dominates} relation or, in case of conflicting access control policies (that simply grant or deny access), simple logical and or or functions.

![Conflict Matrix Diagram](image)

**Figure 6:** The Conflict Matrix

**The Cooperation Matrix.** The cooperation matrix is a concept to organize cooperation between the policy instances within the framework. To that end, the framework contains a partial causal ordering between policy instances via the relation

\[ \text{precedes} \subseteq CI \times CI, \]

In this way an access control policy may be forced to accept only subjects that have been authenticated by one selected authentication policy instance.
The causal *precedes* relation is represented within the framework by the more general *cooperation matrix* $(CI \times CI) \rightarrow CPF$ that maps a pair $i_k, i_l$ to a causal precedence function.

Other examples for the use of causal dependency is ensuring that an encrypted message is decrypted by an algorithm that matches the encryption algorithm, or that message digests are constructed and checked by matching hash functions.

The implementation of both, the conflict and cooperation matrix, is similar to an acl implementation of an access control matrix: the matrices are split into columns, and the columns are associated to the corresponding policy instances.

### A Practical Example: The Chinese Wall Security Policy

This chapter demonstrates the use of the formal framework for describing a security policy in terms of a family language, integrating it into the trusted computing base of a computer system and glueing it together with an application system.

As an example, we chose the Chinese Wall policy [BN89, San92b]. The Chinese Wall policy is a commercial security policy found in companies providing financial consulting services to other, generally competing companies. While the policy allows a market analyst to access the data of companies that are not in competition with each other, the objective of the policy is to prevent information flow that is in conflict with the interests of competing enterprises. The policy guarantees that once an analyst has accessed confidential data about one company he is no longer able to access confidential data of any of the company’s competitors.

The Chinese Wall policy is in several ways well suited for our purpose: the policy has a comprehensible formal definition, it is not focused on military applications alone, it contains both a discretionary and a mandatory component, and it has a dynamic aspect.

To make it fit into the frame of a conference paper, we will use a simplified model closely related to the original version introduced in [BN89]. In contrast to the original model we collapsed all objects in a company dataset into a single object. In contrast to Sandhu’s proposals in [San92b], we omitted the separation of human beings and computing processes. Furthermore, we focused on the *read* rule and omitted the *write* rule. However, the full model including the separation of human beings and computing processes, company object groups, and sanitized information is discussed in [vKO95].

We start with a brief summary of the formal model of the Chinese Wall policy as defined in [BN89]. To allow the reader to easily relate his knowledge about the policy to our example, we use the notation of the original Brewer/Nash paper whenever appropriate.

The entities of the Chinese Wall policy are
(1) a set of subjects $S$ denoting entities that access information contained in objects

(2) a set of objects $O$ denoting entities that contain confidential company-related information

(3) a history matrix

$$N(S, O) \rightarrow \{true, false\}$$

denoting whether a subject has ever accessed an object

(4) an access control matrix

$$A(S, O) \rightarrow P^1\{read, write\} \quad (Power \ set)$$

denoting the discretionary access rights a subject holds with respect to an object

(5) a conflict of interest relation $COI$ between competing companies defining mutually disjoint classes of their corresponding confidential objects. We refer to the conflict class of an object $o$ by $x_o$.

The Chinese Wall read-rule is: any subject $s$ may read an object $o$ if

(1) the discretionary component grants read access

(2) the object belongs to a conflict of interest class within which $s$ has not yet read any other object. Thus, read access of $s$ on $o$ at time $t$ is granted by the rule

$$read \in A_t(s, o) \wedge (N_t(s, o) = true \wedge (\forall o' \in x_o : N_t(s, o') = false)) \Rightarrow N_{t+1}(s, o) = true$$

For a rationale of these rules, see [BN89, San92b].

The Framework’s Algebraic Representation

The following example is an algebraic specification of the Chinese Wall policy. The specification is an element of the set $\mathcal{M}$ (figure 4) and part of the framework’s model level (figure 2).
policy SimpleChineseWall;
import
   Bool, BoolMatrix, ACM, Set, S, O;
types
   R = \{\text{read, write}\}; \quad \% \text{discretionary rights}\%
   Q = \{\text{read, write, none}\}; \quad \% \text{Actions}\%
instantiate
   ACLMatrix is ACM with R as Rights \% generic access control matrix type \%
   ObjectSet is Set with O as Item \% generic set type \%
operations
   COI:\text{ ObjectSet } \rightarrow O
   request:\text{ Q } \times O \times S \rightarrow Q
   DiscrReadCheck:\text{ Bool } \rightarrow O \times S
   MandReadCheck:\text{ Bool } \rightarrow O \times S
   LastSetElem:\text{ ObjectSet } \rightarrow O
   CheckCOI:\text{ ObjectSet } \times S \rightarrow Bool
declare
   N : BoolMatrix; \% \text{History Matrix}\%
   A : ACLMatrix; \% \text{Access Control Matrix}\%
   os: ObjectSet;
axioms
   (1) DiscrReadCheck(o,s) \equiv \text{read} \in A[[o][|s|]]
   (2) MandReadCheck(o,s) \equiv (N[[o]][|s|])=true \lor (\text{CheckCOI(COI(o),s)}=false)
   (3) request(read,o,s)=read \Rightarrow \text{DiscrReadCheck(o,s) } \wedge \text{MandReadCheck(o,s)} \equiv \text{true}
   (4) request\text{(none,o,s)}=\text{none} \equiv \text{true}
   (5) request(request(read,o,s),o,s) \Rightarrow N[[o]][|s|] \equiv \text{true}
   (6) LastSetElem\text{(insert(os,o))} \equiv o
   (7) CheckCOI(os,s) \equiv \text{ite(os = \emptyset, false, or(N[LastSetElem(os)][|s|], CheckCOI(delete(os,LastSetElem(os)),s)))}
requests
   AccessCheck(read,s,o) \models request\text{ (read,s,o)} = \text{read}
end.

Bool, BoolMatrix, ACM, Set, S and O are types imported from the environment. The types, operators, and axioms sections constitute the algebra specification. The declare section declares variables used in the axioms, and requests is the communication interface of the policy.

In the example, we omitted axioms for variables initialization and the (obvious) definitions of imported operations. "\models" denotes "is modelled by", "|s|" uniquely maps
s to its natural number for indexing the matrices. \( \text{ite} \left( \text{boolExpr}, \text{expr2}, \text{expr1} \right) \) is an if-then-else operator modelling conditioned axioms \( \text{boolExpr} \Rightarrow \text{expr1} \equiv \text{expr2} \).

**The Framework’s Expert System Representation**

The true domain of expert system - based security policy development is the incremental development method: the stepwise addition or modification of rules and facts and the rapid prototyping support by the expert system that allows the immediate observation of the consequences.

The following example aims at demonstrating an alternative implementation of the Chinese Wall policy in terms of rules and facts of the Mobal expert system. Evolution and rapid prototyping support is discussed in [KR95]. The specification is an element of the set \( \mathcal{M} \) (figure 4) and part of the framework’s model level (figure 2). The corresponding custodian class additionally contains the Mobal expert system.

In the framework, an expert system representation of a security policy consists of a set of *entities*, a set of *predicates* defining relations between entities, a set of *atomic facts*, and a set of *rules* for generating new facts.

Our Chinese Wall policy example is modelled by the following predicates, rules and facts (in Mobal syntax):

\[
\begin{align*}
A/2 & :< S >, < O > \text{ is a binary predicate implementing the access control matrix} \\
N/2 & :< S >, < O > \text{ is a binary predicate implementing the history matrix} \\
COI/2 & :< O >, < O > \text{ is a binary predicate implementing the conflict-of-interest matrix} \\
\text{can\_read\_discrete}/2 & :< S >, < O > \text{ is a binary predicate implementing the discrete read access permission check} \\
\text{can\_read\_mandatory}/2 & :< S >, < O > \text{ is a binary predicate implementing the mandatory read access permission check} \\
\text{Check\_ReadAccess}/2 & :< S >, < O > \text{ is a binary predicate implementing read access permission check}
\end{align*}
\]

Atomic facts initialize the matrices. As an example, \( \text{COI}(o, p) \) defines a conflict between \( o \) and \( p \), and \( N(s, o) \) denotes that \( s \) once has accessed \( o \). Finally, the following rules complete the model:

\[
\begin{align*}
A(S, O) & \rightarrow \text{can\_read\_discrete}(S, O) \\
\text{unknown}(A(S, O)) & \rightarrow \text{not}(\text{can\_read\_discrete}(S, O))
\end{align*}
\]

\[
\begin{align*}
N(S, O) & \rightarrow \text{can\_read\_mandatory}(S, O) \\
\text{unknown}(N(S, O)) \land N(S, O1) \land \text{COI}(O, O1) & \rightarrow \text{not}(\text{can\_read\_mandatory}(S, O))
\end{align*}
\]
\[
\begin{align*}
\text{not}(	ext{can}_{\text{read}}_{\text{mandatory}}(S, O)) & \rightarrow \text{not}_{\text{mandatory}}(S, O) \\
\text{unknown}(\text{not}_{\text{mandatory}}(S, O)) & \rightarrow \text{can}_{\text{read}}_{\text{mandatory}}(S, O) \\
\text{can}_{\text{read}}_{\text{discrete}}(S, O) \land \text{can}_{\text{read}}_{\text{mandatory}}(S, O) & \rightarrow \text{CheckReadAccess}(S, O) \\
\text{not}(	ext{can}_{\text{read}}_{\text{discrete}}(S, O)) & \rightarrow \text{not}(	ext{CheckReadAccess}(S, O)) \\
\text{not}(	ext{can}_{\text{read}}_{\text{mandatory}}(S, O)) & \rightarrow \text{not}(	ext{CheckReadAccess}(S, O))
\end{align*}
\]

We omitted predicates defining subjects and objects as entities as well as atomic facts initializing the subject and object sets. \(O1\) is a Mobal variable implementing iteration through the object set.

**The Framework Support**

Whenever a security policy is implemented by more than a single model, a choice can be made based on individual model properties. Thus, an expert system model can be chosen as a starting point for the evolutionary development of a new security policy, while algebraic or lattice-based models usually have a better formal verification and performance.

Once a formal model exists, a custodian class \(\in \mathcal{CC}\) must be created. The semantical gap between an algebraic specification and a programming language representation is rather small, so that the transformation of the formal model \(\in \mathcal{M}\) into a custodian class \(\in \mathcal{CC}\) is straightforward [vKO95]. No transformation at all is required by the rules&facts - representation that can be directly interpreted by the Mobal expert system. The corresponding custodian class here contains the expert system interpreter together with the model \(\in \mathcal{M}\).

Once a custodian class exists, an instance of a security policy is created by instantiating the custodian class. By assigning a time interval and a domain, the policy instance is activated. The custodian paradigm then guarantees that the three reference monitor properties hold for the implementation.

Whenever a conflict or cooperation with other security policy instances may happen\(^3\), the new security policy instance is submitted to a metapolicy by entering the instance in the metapolicy’s domain. If several security policy administrators have agreed upon a common metapolicy, submitting a policy instance to a metapolicy can be a mandatory step in the installation of a new policy instance. The metapolicy’s conflict and cooperation matrices then are updated with a new row of conflict resolution and causal precedence functions.

Based on the sets \(\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{M}, \mathcal{D}, \mathcal{CC}\) and \(\mathcal{CI}\) and the operations in \(\mathcal{CRM}\) and \(\mathcal{CPM}\), a future goal will be the definition of an algebra that supports policy comparison and policy synthesizing.

\(^3\)A necessary condition for a conflict between two security policy instances is that the domains of the instances overlap or that application entities within the policy instances domain dynamically interact with applications outside the domain.
Conclusions

The paper gave an overview on a formal framework for supporting multiple security policies. The framework is an emerging security engineering concept, combining reusability concepts similar to those of class libraries of object oriented languages, the metapolicy concept for building bridges between the domains of multiple security policies, and the custodian concept for providing implementation support by transporting the traditional reference monitor properties to user-defined policies.

With the framework in place, the development of a secure application consists of

- the selection of a policy from the framework and the choice of one of its implementations, based on model attributes such as the semantic model description, equivalence relations, performance aspects,

- the creation of a policy instance,

- the definition of instance time and space perimeters,

- the definition of the relations to other policy instances.

Several concepts of the presented work are ongoing research and are yet unproven. However, the success in integrating well-known security policies into the framework has consolidated the soundness of the ideas. The future direction of our work will focus on policy reusability, policy synthesizing and metapolicy support.

Acknowledgment

The authors would like to thank the systems security group at GMD, including Ciaran Bryce, Wolfgang Lux and Mathias Reinholz, for innumerable valuable discussions.

References


